Collaborative Access Control in WebdamLog

Vera Zaychik Moffit, Julia Stoyanovich, Serge Abiteboul, Gerome Miklau

Collaborative Access Control in WebdamLog

Vera Zaychik Moffitt
Drexel University
zaychik@drexel.edu

Julia Stoyanovich
Drexel University
stoyanovich@drexel.edu

Serge Abiteboul
INRIA Saclay & ENS Cachan
serge.abiteboul@inria.fr

Gerome Miklau
UMass Amherst
miklau@cs.umass.edu

ABSTRACT

The management of Web users' personal information is increasingly distributed across a broad array of applications and systems, including online social networks and cloud-based services. Users wish to share data using these systems, but avoiding the risks of unintended disclosures or unauthorized access by applications has become a major challenge.

We propose a novel access control model that operates within a distributed data management framework based on datalog. Using this model, users can control access to data they own and control applications they run. They can conveniently specify access control policies providing flexible tuple-level control derived using provenance information. We present a formal specification of the model, an implementation built using an open-source distributed datalog engine, and an extensive experimental evaluation showing that the computational cost of access control is modest.

1. INTRODUCTION

The personal *data* and favorite *applications* of a Web user are typically distributed across many heterogeneous devices and systems, e.g., residing on a smartphone, laptop, tablet, TV box, or managed by Facebook, Google, etc. Additional data and even computational resources may also be available to the user from relatives, friends, colleagues, and possibly via social network systems. Web users are thus increasingly at risk of leaking their private data and in general of losing control over their own information. In this paper, we introduce a novel *collaborative access control mechanism* that provides users with the means to control access to their data by others and the functioning of applications they run. By collaborative, we mean that different users are willing to participate in the appropriate support of access control.

The focus here is therefore on information management in environments in which data and programs are distributed, either physically between collaborating peers, or conceptually between collaborating principals, i.e., entities that can be authenticated by a computer system or network. (In the following, we will generically use the term "peer".) More precisely, we consider the issue of access control for such an environment. While the technology for distributed information management has been widely studied, access control has rarely been included. In such settings, there are three essential aspects to access control:

R1: Data access As for centralized database systems, users would like to control who can read and modify their information.

R2: Application control Users would like to control which applications can run on their behalf, and what information these applications can access.

R3: Data dissemination Users would like to control how pieces of information are transferred from one participant to another, and how they are combined, with the owner of each piece keeping some control over it.

Finally, motivated by usability and efficiency considerations, we add a fourth, somewhat orthogonal, requirement:

R4: Declarativeness The specification of the exchange of data, applications, as well as that of the access control policies, should be declarative.

To illustrate the first three requirements, let us consider Facebook, a conceptually centralized environment in which millions of users interact by exchanging data and applications. To control who can see her information, a user uses a classic access control mechanism, such as the one currently employed by Facebook, based on groups of friends (**R1**). Next, consider a user installing an application. This typically involves opening much of her data to a server that is possibly managed by an unknown third party. Many Facebook users see this as unreasonable, and would like to control what the application can do "in their name" and what information the application can access (**R2**). Finally, with respect to data dissemination (**R3**), users would like to specify what other users can do with their data, e.g., whether their friends are allowed to show their pictures to their respective friends.

We adopt the last requirement (**R4**) because a declarative specification (i) makes it easier for a user to specify access control policies, (ii) enables reasoning about these policies, and (iii) allows for powerful performance optimizations. The usability consideration is perhaps the most important one, since typical Web users have neither the desire nor the expertise to write code to define how data and applications are exchanged and how access is controlled.

We propose a mechanism that addresses each of these issues by introducing access control in a datalog dialect [4].

1Our work is in scope of the WebdamLog project at Inria [2]. We introduce an access control mechanism into WebdamLog, which is in turn implemented using Bud, an open-source distributed datalog framework [20]. This setting is appropriate for our work because Bud and WebdamLog provide good performance for distributed communication.

Our first contribution is an access control mechanism for WebdamLog. This yields a language that allows the declarative specification of both data exchange and access control policies governing this exchange (**R4**). While a declarative language is not sufficient to guarantee usability, it is an important first step that supports the later design of appropriate interface concepts and the development of GUIs adapted to this setting. Towards (**R1**), we add to WebdamLog an access control mechanism for extensional data that is comparable to existing centralized mechanisms. To support (**R2**), we introduce a mechanism for controlling the use of *delegation* in WebdamLog, that allows peers to delegate work to remote peers by installing rules, and is the main distinguishing feature of the WebdamLog framework. Lastly, to support (**R3**), we introduce a sophisticated fine-grained access control mechanism for intentional data based on *provenance*. To our knowledge, this is the first mechanism that uses provenance for distributed access control.

Our next contribution is an implementation of our collaborative access control model within the WebdamLog system. We improve robustness of the open-source WebdamLog system, implement a sophisticated access control rule rewriting mechanism, and propose interesting performance optimizations aimed at lowering the space overhead and processing time of access control. 1WebdamLog with access control is publicly available and open source at `https://github.com/vzaychik/webdamlog-engine.git`.

Our final contribution is an extensive experimental evaluation, demonstrating that a carefully engineered system can support fine-grained provenance-based access control at a modest cost, which is further reduced by effective performance optimizations. We stress here that we made significant modifications to the state-of-the-art research prototypes on which we build to support our demanding experimental scenarios. Nonetheless, our focus was primarily on showing that the overhead of access control is reasonable.

The paper is organized as follows. We give a brief overview of WebdamLog in Section 2. Our contributions start in Section 3 and Section 4, where we show how WebdamLog can be extended with access control. Section 5 describes system implementation and Section 6 presents the evaluation of performance. We discuss related work in Section 7 and conclude in Section 8.

2. BACKGROUND

In this section, we briefly review WebdamLog [2].

We assume the existence of a countable set of variables and a countable set of data values that includes a set of relation names and a set of peer names. (Relation and peer names are part of the data.) Variables start with $, e.g. x.

Schema. A *relation* in our context is an expression $\mathsf{m@p}$ where m is a relation name and p a peer name. A *schema* is an expression (π, E, I, σ) where π is a possibly infinite set of peer names, E is a set of extensional relations of the form $\mathsf{m@p}$ for $\mathsf{p} \in \pi$, I is a set of intentional relations of the form $\mathsf{m@p}$ for $\mathsf{p} \in \pi$, and σ, the sorting function, specifies for each relation $\mathsf{m@p}$, an integer $\sigma(m@p)$ that is its sort. A relation cannot be at the same time intentional and extensional.[1]

Schema information of peer p is recorded locally in the ternary relation $\mathsf{kind@p}$. For the fact $\mathsf{kind@p}(\mathsf{x_{rel}}, \mathsf{x_{kind}}, \mathsf{x_{arity}})$, $\mathsf{x_{rel}}$ is a relation name, $\mathsf{x_{kind}}$ is its kind, i.e., int or ext (for intentional or extensional), and $\mathsf{x_{arity}}$ is an integer, the arity of the relation. For example, the information that $\mathsf{m_1@p_1} \in E$ and $\sigma(\mathsf{m_1@p_1}) = 3$ is recorded by the fact $\mathsf{kind@p_1}(\mathsf{m_1}, \mathsf{ext}, 3)$. Note that $\mathsf{x_{rel}}$ is the key of relation kind: a relation has a unique arity and kind. For readability, we introduce the following notation. When the fact that a relation is extensional is important, we underline its name. For example, $\underline{\mathsf{kind}}\mathsf{@p}$ is an extensional relation. Names of intentional relations are not underlined. A relation name is not underlined when it is unimportant whether it is extensional or intentional.

Instance. A *fact over a relation* $\mathsf{m@p}$ is an expression of the form $\mathsf{m@p}(\mathsf{a_1}, \ldots \mathsf{a_n})$, where $n = \sigma(\mathsf{m@p})$ and $a_1, \ldots, a_n$ are data values. An example of a fact is:

$\mathsf{pictures@myalbum}(1771.\mathsf{jpg}, \text{“Timbuktu”}, 11/11/2011)$

An *instance* associates a finite set of facts with each extensional relation $\underline{\mathsf{m}}\mathsf{@p}$.

Rules. A *term* is a constant or a variable. (Terms are prepended with $.) A *rule* at peer p is of the form:

$$[\text{at } \mathsf{p}] \quad \$\mathsf{r_0}@\$\mathsf{p_0}(\$\mathsf{U_0}) \text{:- } \$\mathsf{r_1}@\$\mathsf{p_1}(\$\mathsf{U_1}), \ldots, \$\mathsf{r_n}@\$\mathsf{p_n}(\$\mathsf{U_n})$$

where $\$\mathsf{r_i}$ are relation terms, $\$\mathsf{p_i}$ are peer terms, and $\$\mathsf{U_i}$ are vectors of terms. (To simplify, we ignore negative literals in this paper.) The following safety condition is imposed: each variable occurring in the head of the rule (including $\$\mathsf{r_0}$ and $\$\mathsf{p_0}$ if they are variables) must occur in the body. In addition with respect to peer and relation variables, rules are evaluated from left to right and it is required that each $\$\mathsf{p_i}$, $i > 0$, must be previously bound in $\$\mathsf{U_j}$ for some $j < i$.

WebdamLog rule semantics. At a particular point in time, each peer p has a *state* consisting of some facts, some rules specified locally, and possibly some rules that have been delegated to p by other peers. Peers evolve by updating their base of facts, by sending facts to other peers, and by updating their delegations to other peers. So, both the set of facts and the set of delegated rules evolve over time. (To simplify, we follow [2] in assuming that the set of rules specified locally is fixed.) A subtlety lies in the use of variables for peer names. The nature of a rule may depend on the instantiation of its variables, i.e., one instantiation of a particular rule may be local, whereas another may not be.

The semantics of a rule with head $\mathsf{m@p(u)}$ at peer $\mathsf{p'}$ depends on the nature of the relation in its head. It may be extensional ($\underline{\mathsf{m}}\mathsf{@p}$ in E) or intentional ($\mathsf{m@p}$ in I) and local ($\mathsf{p}{=}\mathsf{p'}$) or non-local ($\mathsf{p}{\neq}\mathsf{p'}$). WebdamLog distinguishes further between five main categories of rules, beginning with *local rules*, in which all relations in the body are local:

A. Local rule with local intentional head. These rules, like classical datalog rules, define local intentional predicates, i.e., logical views.

B. Local rule with local extensional head (local database update). These rules derive new facts that are inserted

[1]In deductive database terminology, a fact is *extensional* if it is stored in a database. So an extensional relation is a database relation. A fact is *intentional* if it is derived. Thus an intentional relation is a view.

into the local database. As in Dedalus [6], facts are not persistent by default. To have them persist, one can use rules of the form m@p(U) :- m@p(U). Deletion can be captured by controlling the persistence of facts.

C. Local rule with non-local extensional head. Facts derived by such rules are sent to other peers and stored in an extensional relation at that peer, which allows for a form of messaging between peers.

D. Local rule with non-local intentional head. Such a rule defines a new intentional relation at the remote peer based on local relations of the defining peer.

E. Non-local. Rules of this kind allow a peer to install a rule at a remote peer, which is itself defined in terms of the relations of other remote peers. This is *delegation*, which enables the sharing of application logic by peers. Peers can obtain logic (rules) from other sites and deploy logic (rules) to other sites.

Examples of these rule types will be included in the next two sections, in the presence of access control.

Security assumptions. The existence of a secure communication channel is assumed between any two peers, as well as that of a secure authentication mechanism for peers. Typically, messages between peers will include requests to insert/delete facts and to install rule delegations. The identity of the sender and the receiver of a message is guaranteed by the communication channel.

It is also assumed that peers who are granted privileges on data are trusted to handle those privileges responsibly. As with standard access control, the system cannot prevent improper behavior. A user with read privileges on a data item is not prevented from copying that item, and making the copy available to all peers. An interesting issue that we do not address here is the verification of correct behavior and detection of access right violations. Cryptographic techniques towards those goals have been considered in a related context [3].

The last assumption (for simplicity) is that access control policies are non-sensitive. In particular, each peer can inspect the access rights of other peers in the system.

3. ACCESS CONTROL FOR LOCAL RULES

We next introduce a very rich access control mechanism for WebdamLog allowing the specification of complex access control policies. In this section, we define the semantics of local rules in the presence of such policies. For extensional data, the mechanism we use is standard. Its main originality resides in the use of provenance information for controlling access to *intentional* data. The general mechanism to support READ, WRITE and GRANT in a coherent manner turns out to be rather subtle. To prevent the access control mechanism from becoming too constraining, we introduce means of overriding it in a controlled manner. 1We will demonstrate the practicality and flexibility of our approach in Section 6.

Access control over relational data is called coarse-grained if it is specified at the level of tables or views. Fine-grained relational access control [16, 21, 23] instead operates at the tuple level, or sometimes even at the cell level. We propose a hyprid approach, where users specify coarse-grained access to extensional relations, and the system computes tuple-level annotations for intentional data based on provenance. This approach supports fine-grained access control, while easing the burden of complex access policy specification.

acl@alice(rel, pset, priv)		
friend	{alice, bob}	READ
friend	{alice}	GRANT
photo	{alice, bob, pete}	READ
tag	{alice, bob, pete}	READ
friendPhoto	{alice}	WRITE
allPhotos	{alice, bob}	WRITE

acl@bob(rel, pset, priv)		
friendPhoto	{bob, alice}	READ
friendPhoto	{bob, alice}	WRITE
allPhotos	{bob}	GRANT
friend	{bob}	GRANT

acl@charlie(rel, pset, priv)		
allPhotos	{charlie, alice}	WRITE

Figure 1: Peers alice, bob, and charlie and the relevant portion of their acl relations.

The access control policy is primarily based on a ternary intentional relation acl@p for each peer p that specifies the *access control list* for p — a list of statements defining *privileges* over the relations of p. In a fact $\mathsf{acl@p}(x_{rel}, x_{pset}, x_{priv})$, x_{rel}@p is some relation at p, x_{pset} is a *set* of peers, and x_{priv} is a privilege (to be explained). Its meaning is that each peer in x_{pset} is granted the privilege x_{priv} on relation x_{rel}@p. We consider three privileges:

The READ privilege is required to see the tuples in the specified relation, and applies to both intentional and extensional relations.

The WRITE privilege is required to insert/delete a tuple in an extensional relation. (To simplify, we ignore tuple updates here.) By analogy, the WRITE privilege is required to participate in defining an intentional relation r@p, i.e., adding/removing rules with r@p in the head.

The GRANT privilege is required to grant any privilege on a relation, including GRANT itself, to any peer.

In this section, we consider only local rules. We first informally discuss their semantics under access control (Section 3.1) and then formalize that semantics (Section 3.2). To simplify presentation, we assume in these two sections that the content of acl and kind is given (and fixed). We discuss the specification of the content of these two relations in Section 3.3. We will consider non-local rules in Section 4.

3.1 Access control by example

In this section, we explain by example the effects of access control on the semantics of local rules in WebdamLog. Throughout, we refer to the peers and relations in Figure 1.

Successful execution of a WebdamLog rule in the presence of access control imposes some constraints on the privileges of the executing peer (where the rule is run) and the target peer (the peer occurring in the head of the rule). If the executing peer does not have the WRITE privilege on the relation occurring in the head, the rule has no effect. Assume that the executing peer has such a privilege. For a particular instantiation of the rule, the target peer will receive the resulting tuple only if it has the READ privilege on each fact in the rule body for that particular instantiation.

When WebdamLog rules are executed with access control, one can abstractly think that the rules are "rewritten" with

extra conditions added to the body asserting proper access rights. Indeed, we will formally describe the semantics with access control using such a rewriting in Section 3.2 and this is a core aspect of the implementation. (For instance, we will see that an atom corresponding to "has write privilege on the head relation" will be added to the rule body.) And, as we will see, such a rewriting is also used in the implementation. But for now, we continue with the informal presentation.

Local rules with extensional head. The presence of an extensional relation in the head is interpreted as an insertion into the local database if the head relation is local, or into an external database otherwise. Recall that names of extensional relations are underlined for readability. Consider two rules with the same bodies but different head relations:

[at alice] friendPhoto@alice($ph) :- friend@alice($p), photo@alice($ph), tag@alice($ph, $p)
[at alice] friendPhoto@bob($ph) :- friend@alice($p), photo@alice($ph), tag@alice($ph, $p)

The first rule inserts into a local relation friendPhoto@alice the photos of alice for which at least one of its friends is tagged. This insertion succeeds because alice has READ privileges on all relations in the body and the WRITE privilege on the relation in the head; alice has these privileges because a peer has full access to its own relations.

Now suppose that (i) bob grants the WRITE privilege to alice on friendPhoto@bob and (ii) alice grants READ privileges to bob on all relations in the body. Then the insertion specified by the second rule will succeed. Otherwise it will fail because either (i) alice is not allowed to write or (ii) bob is not allowed to read the resulting tuples.

Local rules with intentional head. Recall that the WRITE privilege on an intentional relation means the privilege to define that relation. Consider two rules defining a relation at alice:

[at alice] allPhotos@alice($f) :- friendPhoto@alice($f)
[at bob] allPhotos@alice($f) :- friendPhoto@bob($f)

The first rule specifies that the local intentional relation allPhotos@alice includes photos from friendPhoto@alice. This rule always executes successfully because alice can define its own relations. The second rule is successful if (i) alice grants the WRITE privilege to bob on allPhotos@alice and (ii) bob grants the READ privilege to alice on friendPhoto@bob. When both (i) and (ii) hold, relation allPhotos@alice contains the union of friendPhoto@alice and friendPhoto@bob.

We make a distinction with respect to access control based on the *kind* (intentional or extensional) of the relation in the head of the rule. Since allPhotos@alice is intentional, access is controlled not only by the relation acl. As for extensional relations, we require the READ privilege on allPhotos@alice for access to the contents of this relation. Additionally, because the relation is intentional, access to it is also controlled by the provenance of each fact. By default, *a peer can access a fact in an intentional relation if that peer's rights are sufficient to derive it.*

If charlie queries allPhotos@alice, it will see the tuples coming from friendPhoto@alice only if it has the READ privilege on friendPhoto@alice; it will see the tuples coming from friendPhoto@bob only if it has READ on friendPhoto@bob. Note that this results in fine-grained access control, since a peer may see only a subset of the tuples in a relation.

Access control for intentional relations is compositional. Consider the following two rules:

[at bob] allPhotos@alice($f) :- friendPhoto@bob($f)
[at alice] allPhotos@charlie($f) :- allPhotos@alice($f)

Without access control, allPhotos@alice and allPhotos@charlie both include all the photos from friendPhoto@bob. In contrast, with access control, allPhotos@charlie contains these photos only if both alice and charlie have the READ privilege on friendPhoto@bob.

Overriding the default semantics. Recall that a rule with an extensional head derives new extensional facts from source facts in the relations in the body. We treat these inferred facts as brand new, with access controlled solely by the acl entry of the extensional relation that contains them. In contrast, rules with an intentional head, which define views that may or may not be materialized, derive facts that retain their relationship with contributing source facts. Access to intentional facts depends on the privileges that allowed their derivation and so is based on provenance information.

We believe this to be the most natural default semantics that will match many common cases. But it should be clear from the examples that this semantics may sometimes be too constraining for intentional data and too liberal for extensional data. This motivates a mechanism for overriding the default semantics, which is achieved by including PRESERVE and HIDE annotations on individual subgoals of rules. Adding HIDE removes provenance history and can be used to override the behavior of intentional rules, making access less restrictive. Adding PRESERVE forces the retention of provenance history and can be used to override the behavior of extensional rules, making access more restrictive. The addition of HIDE is particularly important, because inherited provenance can severely impede information sharing. The HIDE keyword allows for a novel and flexible capability to *declassify* intentional data, as illustrated next.

Hiding provenance for intentional data. Consider a rule that alice uses to publish photos in friends' photo albums:

[at alice] allPhotos@$p($ph) :- [HIDE friend@alice($p)], photo@alice($ph), tag@alice($ph, $p)

Consider $p = pete and suppose that pete is not entitled to read friend@alice. Because of the use of HIDE, photos from photo@alice in which pete is tagged are still copied into allPhotos@pete. In contrast, without HIDE the photos would not be copied. By using HIDE, alice is declassifying some information for pete (namely, that pete occurs in friend@alice) without revealing the entire contents of friend@alice to pete. In this example overriding the default semantics using HIDE prevents access control from becoming too constraining. Another example will highlight a subtlety of HIDE.

Suppose that bob, a friend of alice, re-publishes alice's photos using the following rule:

[at bob] allPhotos@$p($f) :- [HIDE allPhotos@bob($f), friend@bob($p)]

Observe that bob is declassifying data in friend@bob, as he should be entitled to do for his own local extensional relation. However, bob is also declassifying photos of its friends,

e.g., of alice. The question is: Should bob be entitled to do so? The answer is positive only if bob has the GRANT privilege on the corresponding data. This is precisely the semantics enforced by our access control model.

Preserving provenance for extensional data. When an extensional fact is defined by a rule, it does not preserve any information about access constraints on the atoms in the body of the rule that computed it. Because of this a peer can, for example, make a copy of a picture, effectively erasing its provenance record, and subsequently distribute it with no limitations (assuming proper GRANT privileges). To override this behavior, it is possible to tag atoms in the body of a rule with the keyword PRESERVE, with the effect that the provenance for the tagged atoms is kept in the created tuple. For instance, consider the rule:

[at alice] newAllPhotos@bob($ph) :- friend@alice($p),
[PRESERVE photo@alice($ph)], tag@alice($ph, $p)

Suppose that alice intends to prevent charlie from accessing photo@alice, and so withholds READ access on that relation from charlie. By introducing the PRESERVE annotation, alice further makes sure that charlie will not see the photos that were copied into newAllPhotos@bob even if bob grants READ privileges on that relation to charlie.

The annotations PRESERVE and HIDE can be applied to multiple subgoals of a rule. The following example, in which we use two annotations in the same rule, highlights a subtlety of this semantics.

[at p] r@q($x) :- [PRESERVE r_1@p($x)], r_2@p($x)
r@q($x) :- r_1@p($x), [PRESERVE r_2@p($x)]

Although the two rules come from different annotations on a single rule, this yields a disjunction. Suppose that facts r_1@p(a) and r_2@p(a) hold. The fact r@q(a) will hold if q has the READ privilege on either r_1 or r_2.

There is a fundamental difference between the intentional and extensional cases when considering changes. Access to an intentional fact may change as access to the facts from which it was derived changes, much in the same way as its existence depends on the existence of these facts. On the other hand, access to an extensional fact is fixed at the time when it is created, even if the fact was annotated with fine-grained provenance information using PRESERVE. To illustrate, suppose that *after* alice has copied its pictures into newAllPhotos@bob, it grants the READ privilege on photo@alice to charlie. This will not change the accessibility of the tuples already created in allPhotos@bob, and charlie will not see them.

Access control policies. 1An access control policy is (i) a set of rules that define the content of acl@p for each peer p, specifying coarse-grained access, and (ii) HIDE and PRESERVE annotations on rules of a WebdamLog program that define how access is propagated. We will discuss how acl@p is defined in Section 3.3. Formal semantics of a WebdamLog program under a particular access control policy is specified by rewrite rules, which we present next.

3.2 Formal semantics for local rules

The formal semantics of local rules under access control is defined by rewriting the rules of a program P into another set of rules, namely $Acc(P)$, that are evaluated under (almost) standard WebdamLog semantics. Their evaluation enforces the access control semantics.

Although the access control mechanism we introduce can be enforced within the WebdamLog language itself, it turns out that it is convenient to use a language with *set values*. Such values are useful to conceptually capture sets of peers that share the same access privilege. The use of such sets, while natural for the formalization, is essential for performance. We therefore extend WebdamLog, which allows only atomic values, to support set values as well. We do this in the standard manner, so details are omitted. Note that WebdamLog is implemented on top of Bud, which does support nested values. Therefore, from an implementation viewpoint this extension turned out to be straightforward.

In the following, we will use the auxiliary notions of a *p-set*, the predicate *includes*, and *extended relations*:

p-set A *p-set* is simply a set of peers, e.g {alice, bob, charlie}. The p-sets are used to record a particular kind of provenance information that is used for access control.

includes The fact that a peer q is in a p-set P is denoted *includes*(P, q). Observe that *includes* has a universal semantics, so, unlike other relations, *includes* is neither stored extensionally, nor defined by WebdamLog rules.

extended relation For each (intentional or extensional) relation r@p($x_1 \dots x_n$) of arity n, its extended relation, r^+@p($x_1 \dots x_n$, P, PRIV), is of the same kind and has arity $n + 2$.

Coarse-grained access control is recorded in acl relations. For the WRITE privilege, access is controlled at the relation level, i.e., is purely coarse-grained. For example, some q has WRITE access to relation r@p if acl@p(r, P, WRITE) holds for some P that includes q.

For READ or GRANT, access is controlled both at the relation level (coarse-grained) and at the tuple level (fine-grained). Extended relations combine data with access control information to specify access control at the tuple level. For a privilege PRIV that is either READ or GRANT, q will have PRIV access to a fact r@p($a_1 \dots a_n$) if both conditions hold:

coarse-grained q has PRIV access to the relation, i.e., some fact acl@p(r, P, PRIV) holds for some P that includes q;

fine-grained r^+@p($a_1 \dots a_n$, P, PRIV) holds for some P that includes q.

We use the symbol Ω to denote the set of all peers. A fact r^+@p($x_1 \dots x_n, \Omega$, PRIV) indicates that the corresponding fact has no fine-grained access control for privilege PRIV. Also, by definition, we let $P_1 \cap \dots \cap P_k = \Omega$ when $k = 0$.

Note that if r^+ includes two facts r^+@q($x_1 \dots x_n$, P, PRIV) and r^+@q($x_1 \dots x_n$, P′, PRIV) with identical data and privilege values, but with different p-sets, then PRIV on tuple $x_1 \dots x_n$ is available to any peer in P or P', and we replace the two facts by r^+@q($x_1 \dots x_n$, P ∪ P′, PRIV). In the following, we will assume that for a tuple r@q($x_1 \dots x_n$) and a privilege PRIV there is a single corresponding tuple in r^+.

To simplify presentation, we assume next that PRESERVE (respectively, HIDE) annotations appear on the first k (respectively, last $n - k$) subgoals of the n subgoals in the body. Note that since the order of subgoals matters in WebdamLog, this is restricting the language. However, the definitions presented next can be extended to the general case in a straightforward way.

We next consider how READ and GRANT rights are specified combining fine-grained and coarse-grained requirements. For WRITE, such rules are not needed since we assume WRITE is solely governed by coarse-grained access control. We first consider rules with an extensional relation in the head, and PRESERVE annotations.

DEFINITION 3.1 (EXTENSIONAL WITH PRESERVE). Given a local rule with extensional head, assume there are PRESERVE annotations on the first k of n subgoals, for $0 \leq k \leq n$:

$$[\mathsf{at\ p}]\ \underline{\mathsf{r}}@\mathsf{q}(\mathsf{x}_1, \ldots, \mathsf{x}_n)\ \text{:-}\ [\text{PRESERVE}\ \mathsf{r}_1@\mathsf{p}(\mathsf{U}_1)], \ldots [\text{PRESERVE}\ \mathsf{r}_k@\mathsf{p}(\mathsf{U}_k)], \mathsf{r}_{k+1}@\mathsf{p}(\mathsf{U}_{k+1}), \ldots \mathsf{r}_n@\mathsf{p}(\mathsf{U}_n)$$

The rule is rewritten as follows, for PRIV that is either READ or GRANT:

1: $[\mathsf{at\ p}]\ \mathsf{r}^+@\mathsf{q}(\mathsf{x}_1, \ldots, \mathsf{x}_n, \mathsf{F}_1 \cap \ldots \cap \mathsf{F}_k \cap \mathsf{C}_1 \cap \cdots \cap \mathsf{C}_k, \text{PRIV})$:-
2: $\mathsf{r}_1^+@\mathsf{p}(\mathsf{U}_1, \mathsf{F}_1, \text{PRIV}), \ldots \mathsf{r}_k^+@\mathsf{p}(\mathsf{U}_k, \mathsf{F}_k, \text{PRIV}),$
2': $\mathsf{r}_{k+1}^+@\mathsf{p}(\mathsf{U}_{k+1}, \mathsf{F}_{k+1}, \text{GRANT}), \ldots \mathsf{r}_n^+@\mathsf{p}(\mathsf{U}_n, \mathsf{F}_n, \text{GRANT}),$
3: $\mathsf{acl}@\mathsf{p}(\mathsf{r}_1, \mathsf{C}_1, \text{PRIV}), \ldots \mathsf{acl}@\mathsf{p}(\mathsf{r}_k, \mathsf{C}_k, \text{PRIV}),$
3': $\mathsf{acl}@\mathsf{p}(\mathsf{r}_{k+1}, \mathsf{C}_{k+1}, \text{GRANT}), \ldots \mathsf{acl}@\mathsf{p}(\mathsf{r}_n, \mathsf{C}_n, \text{GRANT}),$
4: $\text{includes}(\mathsf{F}_1 \cap \cdots \cap \mathsf{F}_k \cap \mathsf{C}_1 \cap \ldots \cap \mathsf{C}_k, \mathsf{q}),$
4': $\text{includes}(\mathsf{F}_{k+1} \cap \cdots \cap \mathsf{F}_n \cap \mathsf{C}_{k+1} \cap \ldots \cap \mathsf{C}_n, \mathsf{p}),$
5: $\mathsf{acl}@\mathsf{q}(\mathsf{r}, \mathsf{P}', \text{WRITE}), \text{includes}(\mathsf{P}', \mathsf{p})$ □

where each U_i is a tuple of values of the proper arity, and each F_i and each C_j are sets of peers. E.g., $\mathsf{r}_1^+@\mathsf{p}(\mathsf{U}_1, \mathsf{F}_1, \text{PRIV})$ states that peer p knows that each peer in F_1 has PRIV privilege on the tuple $\mathsf{r}_1@\mathsf{p}(\mathsf{U}_1)$.

The justification for this rewriting is as follows:

1. A tuple produced by the head has the p-set resulting from $\mathsf{F}_1 \cap \ldots \cap \mathsf{F}_k \cap \mathsf{C}_1 \cap \ldots \cap \mathsf{C}_k$. It imposes both fine-grained and coarse-grained conditions derived from the source facts that are annotated with PRESERVE. Because the output is extensional, the resulting p-set of each head fact will be frozen after it is created and will no longer depend on its sources.

2,2'. We use extended relations, $\mathsf{r}_i^+@\mathsf{p}(\mathsf{U}_i, \mathsf{F}_i, \text{PRIV/GRANT})$, for each $\mathsf{r}_i@\mathsf{p}$. These provide bindings for the U_i and the fine-grained p-sets F_i.

3,3'. We use p's acl relation $\mathsf{acl}@\mathsf{p}(\mathsf{r}_j, \mathsf{C}_j, \text{PRIV/GRANT})$ for each $\mathsf{r}_j@\mathsf{p}$ in the original rule. These provide bindings for the C_i coarse-grained p-sets.

2,3,4. Because the output tuples are being sent to peer q, we restrict the produced tuples to those on which q has the PRIV privilege. Line (4.) tests that q is in the first k fine-grained and coarse-grained p-sets.

2',3',4'. Because of the declassification of the subgoals (not annotated with PRESERVE), we use the GRANT privilege for these subgoals. Line (4'.) tests that p is in the last $n-k$ fine-grained and coarse-grained p-sets.

5. Finally, peer p must have the WRITE privilege on $\mathsf{r}@\mathsf{q}$ to insert new tuples into $\mathsf{r}@\mathsf{q}$.

In the case where $k = 0$ (i.e., there are no subgoals annotated with PRESERVE), the p-set in the head tuple is the complete p-set, Ω. So, in absence of PRESERVE, no fine-grained access control is imposed on the resulting tuples.

Next, let us consider rules with an intentional relation in the head, and HIDE annotations. Again we consider READ and GRANT rights, combining fine-grained and coarse-grained requirements.

DEFINITION 3.2 (INTENTIONAL WITH HIDE). Given a local rule with intentional head, assume there are HIDE annotations on the last $n-k$ of n subgoals, for $0 \leq k \leq n$:

$$[\mathsf{at\ p}]\ \mathsf{r}@\mathsf{q}(\mathsf{x}_1, \ldots, \mathsf{x}_n)\ \text{:-}\ \mathsf{r}_1@\mathsf{p}(\mathsf{U}_1), \ldots \mathsf{r}_k@\mathsf{p}(\mathsf{U}_k), [\text{HIDE}\ \mathsf{r}_{k+1}@\mathsf{p}(\mathsf{U}_{k+1})], \ldots [\text{HIDE}\ \mathsf{r}_n@\mathsf{p}(\mathsf{U}_n)]$$

The rule is rewritten, for PRIV that is either READ or GRANT exactly as (1-5) in Definition 3.1. □

The justification for the rewriting is virtually identical to the extensional case. The first k subgoals are treated similarly by default and the remaining subgoals are explicitly annotated, but with HIDE. Because the head of the rule is intentional, the derived facts reflect, at all times, changes in the facts that produced them (including their p-sets), whereas these were frozen for extensional facts.

In the case when $k = 0$ (i.e. there are no subgoals annotated with HIDE), the facts in the head inherit fine-grained and coarse-grained annotations from all facts in the body, which is the default semantics for an intentional-head rule.

Note that if $q = p$ (in the previous two definitions), then line 5 of the rule in Definition 3.1 always holds.

Finally, observe that both extensional and intentional facts have two extra columns in extended relations. An issue arises when starting a WebdamLog program from an initial state with nonempty extensional relations that are lacking these two extra columns. To initialize the extended extensional relations, the system executes the following rule for each $\underline{\mathsf{r}}@\mathsf{p}$:

$$[\mathsf{at\ p}]\ \underline{\mathsf{r}^+}@\mathsf{p}(\mathsf{x}_1 \ldots \mathsf{x}_n, \Omega, \text{PRIV})\ \text{:-}\ \underline{\mathsf{r}}@\mathsf{p}(\mathsf{x}_1 \ldots \mathsf{x}_n)$$

In this case, the facts in $\underline{\mathsf{r}^+}@\mathsf{p}$ are created with no fine-grained constraints: all p-sets are Ω. The constraints specified by the acl relation will be applied in the above rewriting whenever $\underline{\mathsf{r}^+}@\mathsf{p}$ is used.

3.3 Defining acl and kind

The relation $\underline{\mathsf{kind}}@\mathsf{p}$ is extensional, whereas, as previously mentioned, the relation $\mathsf{acl}@\mathsf{p}$ is intentional. The main reason for this is to have more flexibility in specifying access control. Consider the following two rules:

[at alice] acl@alice(photos, GRANT, bob) :-
[at bob] acl@alice(photos, READ, $x) :- friends@bob($x)

With the first rule, bob is allowed by alice to grant privileges on photos@alice. With the second, bob grants the READ privilege on photos@alice to all its friends. Note that bob can do this only because it was granted the GRANT privilege by alice.

Observe how the content of friends@bob impacts access control on photos@alice. In particular, observe how unfriending someone (i.e., deleting some tuple from relation friends@bob) results in revoking access rights.

Now consider the following two rules:

[at alice] acl@alice(acl, GRANT, sue) :-
[at sue] kind@alice(newPhotos, int, 3) :-

With the first rule, sue receives (from alice) all privileges to alice's peer, allowing sue to create and delete relations.

Now, with the second rule sue introduces a new relation newPhotos@alice. By default, both alice and sue have the GRANT privilege on the newly created relation, and so they also have READ and WRITE on this relation.

We use the following rules to enforce some basic constraints on the acl relations and to control access to the acl relations themselves.

- If p has the GRANT privilege on r@q, then p also has READ and WRITE privileges on r@q. (Note that p grants these privileges to itself.)
- Peer q has the GRANT privilege on each relation in q and, in particular, on acl@q. Thus q can create relations and define access policies on its own peer.

These constraints can be expressed using WebdamLog rules and can easily be enforced by the engine. Note that the GRANT privilege provides a fine-grained administration mechanism, in that p can give q administrative access to its relations by granting GRANT on acl@p to q. Our framework does not currently support roles, or any other type of administration mechanism beyond that of the GRANT privilege.

We conclude with a remark on access control on the acl relation. Observe that the WRITE privilege on relation acl provides the GRANT privilege on all relations of the corresponding peer. Note also that one can control READ access to an acl relation like to any standard relation. The fact that some peer does not have access to such relations further limits the distribution of information.

3.4 Complexity

Finally, we consider the issue of complexity. A natural question to ask is whether the introduction of access control increases the complexity of accessing information. In [2], the authors show that for positive and local WebdamLog programs, one can compute the final state and answer queries in PTIME data complexity (the program is assumed to be fixed and the complexity is in the size of the input instance). The proof follows from the fact that, as in datalog, the number of tuples that can be derived in a computation is polynomial in the size of the input instance. The proof does not carry immediately to WebdamLog with access control because the p-sets we use could possibly introduce an exponential blow-up in the number of facts that may be derived. However, recall that we replace two facts in $r^+@q$ that agree on data and PRIV values and have p-sets P and P' by $r^+@q(x_1 \ldots x_n, P \cup P', \text{PRIV})$. It then turns out that the number of facts remains polynomial, because the atomic attributes of each relation form a key for that relation, as in V-relations [4]. Thus, for positive and local WebdamLog programs with access control, one can compute the final state and answer queries also in PTIME. Details are omitted. Thus, from the theoretical viewpoint, our access control mechanism does not increase the complexity for positive and local programs beyond PTIME. Indeed, experimental results in Section 6 will demonstrate that a carefully engineered implementation of our rich access control mechanism does not significantly degrade performance.

4. NON-LOCAL RULES

General delegation is based on rules with non-local relations in the body and is the main distinguishing feature of WebdamLog. With delegation, a peer p can ask another peer q to do processing on its behalf. Delegation provides significant flexibility for application development but also raises challenges for access control.

The following example illustrates the danger of a simplistic semantics for non-local rules. Consider two rules:

[at bob] message@sue("I hate you") :- date@alice(d)
r@bob(x) :- date@alice(d), secret@alice(x)

Without access control these two rules are installed at alice. Assuming date@alice(d) succeeds, alice will send hate mail to sue as a result of the first rule. Next, as a result of the second rule, the entire relation secret@alice will be copied into r@bob, even if alice did not intend to give bob access to this data. The main reason for these problem is that, by the standard semantics of WebdamLog, the rules delegated to alice by bob are executed *as if they were originated by* alice, which is clearly unacceptable.

With access control, we are going to run these two rules at alice in a *sandbox* with bob's privileges. For the first rule, the hate message will be sent (assuming that bob has GRANT on date@alice and WRITE on message@sue) but marked as coming from bob. For the second rule, data will be sent only if bob also has READ or GRANT access to secret@alice.

For a client bob delegating a rule to a server, the semantics of delegation under access control guarantees that:

- If the rule has side effects, e.g., it results in the insertion of tuples into some relation of sue, these side effects are attributed to bob.
- The rule executes with bob's access privileges.

Note that, in practice, when alice sends sue a message saying that the author of the message is bob, sue may question this fact and ask alice for a proof. The delegation from bob to alice can be presented to sue and serve as such a proof.

A natural question to ask is whether delegating processing from p to q, with access control that uses sandboxing, will yield exactly the same semantics, with possibly different performance, as if p were getting the data locally and running a local computation. It turns out that this is not the case. This is because, when evaluating p's delegation, q will use data (i) to which it has access (by definition q cannot access data to which it has no access) and (ii) to which p has access (because of sandboxing). On the other hand, in a local computation p is only limited by (ii) but not by (i).

We conclude this section by mentioning a more permissive way of handling access control in delegation. The semantics we described in this section is very protective and typically results in testing extra conditions. A local rule running at p can read p's data, whereas if it were running at p on behalf of q it would be necessary to check that q has the READ privilege on the data. In some cases, p may choose to give q full privileges to run rules at p as if these rules were installed by p itself. For example, this may be the case if q is the smartphone of user Alice and q is her laptop.

5. SYSTEM IMPLEMENTATION

In this section, we describe the implementation of our collaborative access control model in the context of the WebdamLog system. We started from the implementation of the WebdamLog system and extended it to support access control. WebdamLog is implemented on top of the Bud system, an open-source distributed datalog engine with updates and

asynchronous communication [5]. All processing is done in memory and all intentional relations are materialized.

Each WebdamLog peer processes its program in ticks, made up of 4 parts. In Part 1, a peer receives messages from other peers, and acts on the messages by adding or deleting facts, installing or removing collections, and installing or removing rules. Modifying a peer's program arises due to delegation and is specific to WebdamLog. In Part 2 of a tick, a peer's program is *re-wired* in response to any changes from Part 1, i.e., the dependency graph is re-computed and relations are invalidated as appropriate. In Part 3, a peer's program is run to fixpoint. In Part 4, outgoing messages are created and sent to other peers. In our experiments, we report time until fixpoint (sum of Part 3 for all ticks), which corresponds to time spent by Bud in standard database computation, and total tick time (sum of Parts 1 through 4), which corresponds to the total time of computation by Bud and WebdamLog.

5.1 Implementation of access control

We improved the implementation of WebdamLog to make processing more robust so as to support our demanding experimental scenarios. We now describe some of these improvements. Message semantics differ between Bud and WebdamLog. In Bud, each message is expected to be a single tuple, the order of messages is not important, and message loss is acceptable. Bud natively supports the UDP protocol for peer-to-peer communication. In WebdamLog, message loss is not acceptable because messages are not resent. We worked together with Bud developers to add TCP messaging to Bud. We also implemented message queuing to buffer messages intended for peers not yet started.

Recall from Section 3 that access to the extensional relations of peer p is coarse-grained, and is specified in acl@p. A relation $\underline{x_{rel}}$@p appears in three tuples: acl@p(x_{rel}, x_{pset}, READ), acl@p(x_{rel}, x_{pset}, GRANT), and acl@p(x_{rel}, x_{pset}, WRITE). The PUBLIC access policy, in which a relation is accessible to all peers, is specified by a special p-set symbol Ω.

Further, recall from Section 3.2 that information in acl@p is used in rule evaluation to derive fine-grained (tuple-level) READ and GRANT access to some relation r@p, with the resulting p-set values recorded in a column of the corresponding extended relation r^+@p($x_1 \ldots x_n, x_{pset}$, PRIV). This is accomplished by rewriting every WebdamLog rule as per Definitions 3.1 and 3.2, repeated below.

1: [at p] r^+@q($x_1, \ldots, x_n, F_1 \cap \ldots \cap F_k \cap C_1 \cap \cdots \cap C_k$, PRIV) :-
2: r_1^+@p(U_1, F_1, PRIV), $\ldots$ r_k^+@p(U_k, F_k, PRIV),
2': r_{k+1}^+@p(U_{k+1}, F_{k+1}, GRANT), $\ldots$ r_n^+@p(U_n, F_n, GRANT),
3: acl@p(r_1, C_1, PRIV), $\ldots$ acl@p(r_k, C_k, PRIV),
3': acl@p(r_{k+1}, C_{k+1}, GRANT), $\ldots$ acl@p(r_n, C_n, GRANT),
4: includes($F_1 \cap \cdots \cap F_k \cap C_1 \cap \ldots \cap C_k$, q),
4': includes($F_{k+1} \cap \cdots \cap F_n \cap C_{k+1} \cap \ldots \cap C_n$, p),
5: acl@q(r, P′, WRITE), includes(P′, p) □

To make processing more efficient, we create two extended relations, r^+_{read}@p($x_1 \ldots x_n, x_{pset}$) and r^+_{grant}@p($x_1 \ldots x_n, x_{pset}$). Because of this, we must now rewrite each WebdamLog rule into two access control rules, one for READ and one for GRANT. Nonetheless, storing two extended relations rather than one with twice as many tuples proved more efficient in our experiments, leading to an improvement of about 40% in the fixpoint computation on average.

aclf@alice(rel, sym, priv)

birds	A	READ
art	B	READ
fave	C	READ

formula@alice(sym, expr, pset)

A		{Bob, Cathy, Don}
B		{Cathy, Ezra}
C		{Bob, Ezra}
D	A ∩ C	{Bob}
E	B ∩ C	{Ezra}
F	D ∪ E	{Bob, Ezra}

album+@alice(ph, sym, priv)

a101.jpg	D	READ
a102.jpg	F	READ
a104.jpg	E	READ

birds@alice(ph)

a101.jpg
a102.jpg

art@alice(ph)

a102.jpg
a103.jpg
a104.jpg

fave@alice(ph)

a101.jpg
a102.jpg
a104.jpg

Figure 2: Example of Optimization 2 (formulas)

5.2 Performance optimizations

Writeable (Optim 1). Consider again the rewriting given in Definition 3.1. In the basic implementation of access control, p does not have any information about its own WRITE privileges on relations of a remote peer q. To evaluate the rule, p sends tuples to a temporary relation on q and delegates a rule to push these tuples into r@q, subject to acl@q(r, P′, WRITE), includes(P′, p) (line 5 of the rule).

To avoid sending tuples that may be rejected, we check the condition acl@q(r, P′, WRITE) and includes(P′, p) locally at p. This is done by adding a relation writeable@p(r, q) that contains a tuple for each remote relation r@q for which p has the WRITE privilege. In our implementation, writeable@p(r, q) is loaded from the file system to simulate the persistent nature of peers. This relation can also be populated on start-up based on information that is exchanged between peers.

Formulas (Optim 2). When evaluating access control rules, the system exchanges p-set values between peers and intersects them as part of rule evaluation. Long p-sets can arise naturally in this process, and significantly increase message size (i.e., communication cost) and rule evaluation time. To address these issues we observe that, since fine-grained access is derived by a small number of rules from a small number of base relations, to which access is coarse-grained, it is likely that there are relatively few distinct p-sets. The formulas optimization is based on this observation.

The idea is to associate a symbol with each unique p-set that occurs among annotations of tuples on a peer p. Figure 2 illustrates our approach for the program:

```
[at alice]
album@alice($ph) :- birds@alice($ph), fave@alice($ph)
album@alice($ph) :- art@alice($ph), fave@alice($ph)
```

We maintain a relation formula@p(sym, expr, x_{pset}) that associates a symbol with a formula and with the resulting p-set. The contents of formula@p are computed on the fly during rule evaluation, and are maintained independently for each peer. Each formula symbol used in tuples sent to a remote peer is included in that message, together with the corresponding p-set, but without the deriving expression. Observe that symbols are used instead of p-sets in the formula-specific version of the acl relation called aclf@alice and, consequently, in the tuples of extended relations like album$^+$@alice. In this example, we store (and look up) the

symbol F and the associated peers {bob, ezra} that have READ access to "a102.jpg". Without this optimization, this set would need to be recomputed as: {{bob, cathy, don} ∩ {bob, ezra}} ∪ {{cathy, ezra} ∩ {bob, ezra}}.

Formulas act as a cache of common p-set expressions. Using formulas comes at the cost of maintaining additional data structures, and of making rule rewriting more complex. Nonetheless, as we will demonstrate in Section 6, this optimization significantly improves performance because it allows us to avoid recomputation of long complex p-sets.

6. EXPERIMENTAL EVALUATION

1The access control mechanism we propose is flexible and can be tailored to the needs of particular applications. Access control rewriting of Section 3.2 will introduce overhead. The more fine-grained the policy, the higher the overhead of the policy is expected to be. The goal of the experimental evaluation is to demonstrate that this overhead is modest even for the most demanding policies.

We evaluate the performance of our implementation using two realistic data exchange scenarios. The first, "Photo Album" (PA), implements a variant of our running example using networks of peers extracted from Facebook. The second, "Master-Aggregators-Followers" (MAF), is fully synthetic and allows studying the effect of database size and network size/topology on performance. The scale of our experimental evaluation is in-line with our motivating application, personal information management. We target queries involving a number of nodes that is in the hundreds, corresponding to the size of an average user's social network. 1In both scenarios, each peer is running under its own access control policy that is distinct from other peers. We further vary policies by access control conditions, described later in this section.

Scenario 1: Photo Album (PA). In this scenario, we execute a WebdamLog program that computes the contents of a photo album featuring photos in which alice and bob appear together. This computation uses two salient features of WebdamLog, namely, peer variables and delegation, and is made up of two steps. First, names of peers from which photos will be retrieved are gathered in allFriends@sue($peer) by taking a union of friend@alice($peer) and friend@bob($peer).

```
[at sue] allFriends@sue($peer) :- friend@alice($peer)
         allFriends@sue($peer) :- friend@bob($peer)
```

Next, sue gathers photos from remote peers by retrieving photos in which both alice and bob are tagged.

```
[at sue] album@sue($photo, $peer) :-
           allFriends@sue($peer), photo@$peer($photo),
           tag@$peer($photo, alice), tag@$peer($photo, bob)
```

This program is executed over 20 sample networks of varying size (from 20 to 250 peers), which are built using a historical crawl of Facebook. To generate these networks, we randomly sample a pair of nodes connected by a friendship edge to represent alice and bob and use the nodes in their immediate neighborhood to represent their friends.

Each photos@$peer contains about 1,000 facts represented by integer values. Tags are assigned to photos independently for each {$peer, $photo} pair, with the probability of tagging a photo with alice or with bob set to 10%, and probability of tagging a photo with any other peer name set to 1%. The resulting size of album@sue($photo, $peer) with no access control, or, equivalently, under the PUBLIC access control policy, varies depending on network size, and is between 100 and 1,600 tuples. We will discuss access control policies later in this section.

Scenario 2: Master-Aggregators-Followers (MAF). In a MAF network, one master peer gathers data from n aggregators (agg), which in turn gather data from a total of m followers (fol), with k aggregators per follower (described as MAF $n/m/k$). For example, in a MAF (10/2/1) network, there are 10 followers and 2 aggregators, with 1 aggregator per follower and $\frac{10}{2} = 5$ followers per aggregator.

In our experiments, we vary n, m and k to investigate the effect of network size (number of peers) and network topology (number of aggregators per follower) on execution time and space overhead.

We experiment with two flavors of the MAF scenario. In a union of joins (UoJ), each aggregator takes a join of data from the relevant followers and the master then takes a union of results from aggregators. For example, the following program is executed in MAF(3/3/2)-UoJ.

```
[at master] t@master($x) :- s@agg1($x)
            t@master($x) :- s@agg2($x)
            t@master($x) :- s@agg3($x)
            s@agg1($x) :- r@fol1($x), r@fol3($x)
            s@agg2($x) :- r@fol1($x), r@fol2($x)
            s@agg3($x) :- r@fol2($x), r@fol3($x)
```

In a join of unions (JoU), aggregators take a union of follower data and master joins results from the aggregators. The following program is executed in MAF(3/3/2)-JoU.

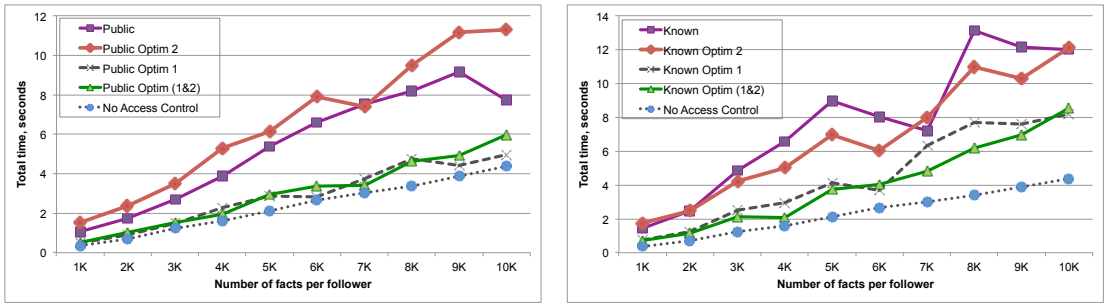
```
[at master]
   t@master($x) :- s@agg1($x),s@agg2($x),s@agg3($x)
   s@agg1($x) :- r@fol1($x)      s@agg1($x) :- r@fol3($x)
   s@agg2($x) :- r@fol1($x)      s@agg2($x) :- r@fol2($x)
   s@agg3($x) :- r@fol2($x)      s@agg3($x) :- r@fol3($x)
```

Note that in both MAF-UoJ and MAF-JoU, rules are initially installed on master and then delegated to agg and fol peers. This is done to make timing of experiments easier and is not essential to the computation.

Each follower peer's extensional relation r@fol(x) is populated with randomly generated facts. We vary the number of facts from 1,000 to 10,000 in increments of 1,000, and restrict the domain of $x to between 1,000 and 10,000 distinct integer values for the respective cases.

Scenarios PA and MAF differ in three important ways. First, the flow of information is different: MAF is a 3-level pyramid, with information flowing from the bottom to the top, while in PA sue first requests all data and then collects it. Second, the program in PA is more complex, making use of peer variables and self-joins. Third, the number of messages being exchanged is much higher in PA, resulting in a higher number of fixpoint computations — about 250 at sue for the largest PA network, compared to about 20 at master for the largest MAF network.

Access control conditions. Each PA and MAF scenario is executed under three different conditions: access control disabled, a PUBLIC access control policy, and a KNOWN access



(a) PUBLIC access control policy (b) KNOWN access control policy

Figure 3: Total time, averaged across agg peers in MAF-JoU (10/2/1), as a function of database size.

control policy. Under the PUBLIC policy, all peers have READ access to all extensional relations of other peers. This policy computes the same result in album@sue($photo, $peer) (for PA) and t@master(x) (for MAF) as when the WebdamLog program is executed without access control. Comparing performance under the PUBLIC policy to performance without access control gives an understanding of the overhead introduced by the relations required to maintain access control information and by access control rule rewriting.

Under the KNOWN policy, peers have READ access to the extensional relations of the peers to which they are connected by a direct edge. This policy represents a common case in social networks such as Facebook, where one's friends have access to their data but peers outside the friendship network do not. This policy allows us to evaluate the overhead of computing non-trivial p-sets, and of propagating them through access control rule rewriting. A program under the PUBLIC policy is more efficient than under the KNOWN policy, provided that it computes the same result under both policies. This is because PUBLIC access to a fact is represented compactly by the symbol Ω, rather than by a variable-length p-set required for the KNOWN policy.

A subtlety in evaluating the KNOWN policy arises in the MAF-UoJ scenario, due to delegation in WebdamLog being processed left-to-right. Consider the following rule.

[at master] s@agg($x) :- r@fol1($x), r@fol2($x), r@fol3($x)

This rule is evaluated by installing the following chain of delegations on participating peers.

[at fol1] temp@fol2($x) :- r@fol1($x)
[at fol2] temp@fol3($x) :- temp@fol2($x), r@fol2($x)
[at fol3] s@agg($x) :- temp@fol3($x), r@fol3($x)

For a non-empty result to be computed in s@agg, fol2 must have READ access to r@fol1, and fol3 must have READ access to r@fol1 and r@fol2. To simplify policy generation, we assume all fol peers that send data to the same aggregator have READ access to each others r relations. An analogous situation arises in MAF-JoU, when a join of relations at each aggregator is processed left-to-right. To ensure that a non-empty result is computed in t@master, we grant READ access to each agg peer on all r relations of fol peers.

Remark. We did not run any experiments with policies that include HIDE and PRESERVE. These constructs make the policy language more flexible, but do not add new implementation challenges: evaluating under HIDE is identical to extension evaluation; evaluating under PRESERVE is identical to intentional evaluation.

Experimental environment. All experiments are conducted on a cluster of 8 Linux nodes running CentOS 2.6.32 (64-bit). Six cluster nodes have Quad-core Intel(R) Xeon(R) CPU X5460 @ 3.16GHz with 8G of RAM. Two other cluster nodes have 16 Intel(R) Xeon(R) CPU E5-2643 0 @ 3.30GHz with 15.65G of RAM.

For the PA scenario, peers sue, alice and bob share a single node, while other peers are spread out evenly across the remaining nodes. For the MAF scenarios, master runs on a dedicated node and the remaining peers are spread out evenly, with agg and fol peers running on separate nodes.

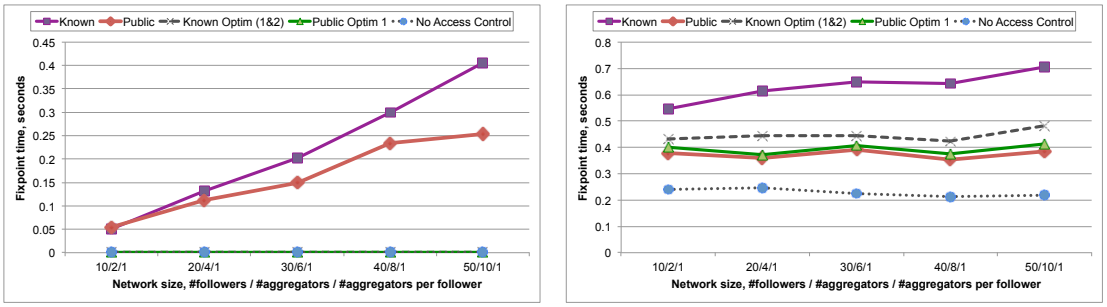
All experiments are executed 5 times with a cold start. We report averages of 5 executions.

Performance optimizations. We start by presenting a comparison of different access control conditions with and without performance optimizations, using a representative example. Figures 3(a) and 3(b) show the average running time of agg peers for the JoU (10/2/1) scenario as a function of database size (namely, size of r@fol at each fol). In this scenario, an agg peer takes a union of extensional relations from 5 fol peers.

Observe from Figure 3(a) that Optim 1 (writeable) significantly outperforms the unoptimized implementation of PUBLIC access control, while Optim 2 (formulas) performs slightly worse than the unoptimized version. Finally, the combination of Optim 1 and Optim 2 performs comparably to Optim 1 alone and to no access control. (We denote this combination by Optim (1 & 2).) These results are as expected. Formulas do not speed up the running time for PUBLIC because we store p-sets for this policy using the special symbol Ω, which is already as compact a representation of a p-set as can be. In fact, adding formulas introduces another level of indirection: a relation is added to the rewriting and rewriting rules become more complex, without the benefit of making p-sets more compact.

Next, observe from Figure 3(b) that Optim 1 (writeable) significantly outperforms the unoptimized implementation of the KNOWN policy, and that Optim 2 (formulas) alone slightly outperforms the unoptimized implementation. The best results are achieved with Optim (1 & 2) because p-sets are non-trivial for the KNOWN policy, and formulas make working with p-sets more efficient during fixpoint computation and during data exchange between peers.

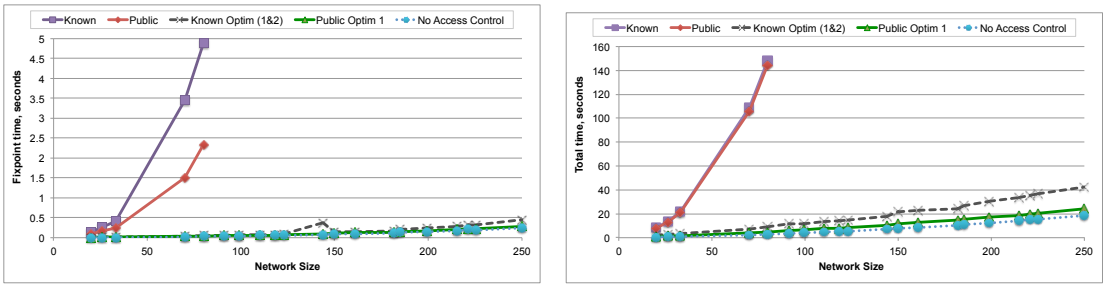
We observed similar trends in all scenarios in our experiments, for all types of peers. In some cases, Optim (1 & 2) performs slightly worse than Optim 1 alone for the PUBLIC policy, but in most cases performance of these two options is



(a) MAF-UoJ, time at master

(b) MAF-UoJ, average time at fol peers

Figure 4: Fixpoint time in MAF-UoJ as a function of network size.



(a) fixpoint time

(b) total time

Figure 5: Running time of peer sue in PA, as a function of network size.

comparable, and is significantly better than performance of the unoptimized version. Optim (1 & 2) is always the most efficient option for the KNOWN policy, and is significantly more efficient than the unoptimized version.

In summary, the PUBLIC policy under Optim 1 achieves performance comparable to no access control, and so is as efficient as can be. We use this configuration in our experiments in the remainder of the section. The best performance for KNOWN is achieved by Optim (1 & 2), and we use this configuration in our experiments. As expected, access control does introduce overhead, but it is modest.

Varying network size. We fix the size of each r@fol to 10,000 facts and vary the number of agg and fol peers, while keeping the ratio of followers to aggregator constant at 5.

Figures 4(a) and 4(b) show the running times to fixpoint with and without access control for MAF-UoJ scenarios. The overhead of optimized access control is either fixed for the optimized implementations or shows a slight linear slope. The running time of agg peers was under 1.5 sec (total) in this scenario and we omit these plots. (Figure 7 in the appendix presents total running times corresponding to the fixpoint times in Figure 4.) Similar trends are observed in the MAF-JoU, see Figures 6(a) and 6(b) in the appendix.

We now consider the photo album (PA) scenario, which we execute with 20 networks of varying size, fixing the size of all photos@$friend relations to about 10,000 facts. Figure 5 shows the running time to fixpoint and the total time on peer sue. We do not complete executing unoptimized versions of the program with PUBLIC and KNOWN policies for networks with more than 80 peers because a significant portion of the running time in the unoptimized PA scenario is spent in Part 2 of the tick (see Section 5), which re-wires the Bud program in response to program updates. Majority of the time is spent on dependency graph computation, which appears to be exponential in the number of relations and dominates the total running time for networks of more than 50 peers. These effects are amplified by the limited amount of RAM per peer in our experiments, especially for larger networks, where we run up to 30 peers per cluster node.

The optimized version of our implementation is able to handle networks with up to 250 peers using our experimental environment (8 physical cluster nodes), and shows reasonable performance, with both fixpoint time and total time increasing linearly for all access control conditions (including no access control). The linear increase is due to the direct relationship between the size of the network and the number of ticks executed by sue, since WebdamLog processes each peer message separately. Figure 8 in the appendix presents fixpoint and total times on peer friend, averaged across peers.

We also experimented with the effect of network topology (number of aggregators per follower) for the MAF scenario. Figure 9 in the Appendix presents these results.

In summary, the overhead of access control increases with increasing network size but remains reasonable for the optimized implementations. The overhead of access control grows at most linearly with increasing network size for MAF and PA scenarios (for the optimized versions) and quadratically for PA for the unoptimized versions.

Space overhead of access control. In the final set of experiments, we consider the space overhead introduced by access control. We focus on the MAF-JoU (10/2/1) scenario and vary the size of r@fol from 1,000 to 10,000. Figure 10 in the appendix presents these results, and is followed by a brief discussion. Access control introduces a modest amount of space overhead, but performance optimizations are effective at reducing this overhead.

7. RELATED WORK

Bud [5] is a declarative datalog-based language implemented in Ruby. Our implementation uses Bud as a transport mechanism and distributed datalog engine. We made a number of enhancements to Bud, described in Section 5.

The WebdamLog language was first formally described in [2] as a version of distributed datalog in which peers exchange not only facts, but also rules. Expressiveness and semantic issues were formally investigated, but access control was not considered. Methods for enforcing basic access control primitives over a distributed data model are considered in [3]. The authors study cryptographic techniques to support, for example, authentication. The techniques they propose can be combined with those we present here.

Controlling access to intentional facts in WebdamLog is related to managing virtual views in SQL, which is handled differently among current database systems. When an authorized user accesses a view, it is usually evaluated with the privileges of the defining user ("definer's rights") although some systems (e.g. mySQL) permit access to views using the privileges of the invoker of the view ("invoker's rights"). Our model supports a novel set of alternatives that encompasses both invoker's and definer's rights.

The access control model we have described is fine-grained, unlike the SQL standard. Lefevre et al. [16] proposed a fine-grained access control model for implementing personal privacy policies in a relational database. They use query modification to enforce their policies, as we do, but their policy model and implementation assume a centralized database system. A commercial example of fine-grained access control is Oracle's Virtual Private Database (VPD), which supports access control at the level of tuples or cells. Alternative semantics for fine-grained access control have been investigated thoroughly [16, 21, 23]. Rizvi et al. [21] distinguish between Truman and Non-Truman models. (The expression is motivated by the movie *The Truman Show* where the hero is unaware that he lives in an artificial environment.) Query answers in our system follow the Truman paradigm: queries are not rejected because of lack of privilege but the user's privileges determine the answers that are returned.

Our model of access control shares some features with the model of reflective database access control (RDBAC) in which access policies can be specified in terms of data contained in any part of the database. Olson et al. [17] formalize RDBAC using a version of datalog with updates [9] but their model does not include distribution, delegation, or the use of provenance.

The use of provenance as a basis for access control was first noted in the context of provenance semirings [14, 7]. A security semiring can contain tuple-level security annotations and define the rules by which they are propagated to query results. Another example of provenance-based access control is the work of Park et al. [19] in which access decisions are based on a transactional form of provenance.

The emergence of social networks and other Web 2.0 applications has led to the adaptation of access control to these domains. In online social networks, the distinguishing feature of access control models is that a policy is expressed in terms of network relationships amongst members [10, 13]. A number of advanced features for access control in online social networks have also been studied. For example, Hu et al. [15] focus on enabling multiple parties to simultaneously control access to resources, including a voting scheme to resolve conflicts. Cheng et al. [12] emphasize policies expressed over diverse relationships (e.g. resource-user and resource-resource relationships, not just user-to-user relationships). Shehab et al. [22] focus on controlling the accessibility of data by third-party applications, including automatic generalization of personal information. Our framework can support arbitrarily complex relationships, since our data model is not fixed, but we do not focus on multiparty control and we do not consider transformations of the data when it is deemed accessible to a peer. A number of policy specification languages have been proposed, including fine-grained access control based on semantic web representations of the social network and authorizations, allowing existing tools like SWRL and SPARQL to be deployed for policy enforcement [11]. Each of these works assumes a centralized implementation of the social network; although the access control models may in some cases extend to a distributed setting, their implementation does not.

We emphasize that while we used a social network as a motivating example and for our experimental setting, the core of our framework is much broader and is intended to support the diverse requirements of access control in a variety of distributed information-sharing applications.

Also, security in distributed systems has primarily focused on issues of remote authentication, authorization, and protection of data and distributed trust; such issues are outside the scope of our present work [1, 18].

8. CONCLUSION

We have described a novel access control model for distributed data management that allows peers to declaratively specify powerful policies governing access to their data, dissemination of their data, and delegation of computation. Building upon the distributed data processing and transport mechanism provided by Bud and the high-level language constructs of WebdamLog, we implemented our access control model, proposed key performance optimizations, and demonstrated that performance overhead of access control evaluation is modest in realistic scenarios.

In the future we would like to investigate automated tools for reasoning about policies, which can help peers understand the consequences of policies they author and detect policy errors. While computationally intractable in the worst case, we hope to find useful, practical constraints that lead to efficient policy analysis methods. Another feature that we do not currently support, but plan to investigate in the future, is allowing users to iteratively refine access control policies. This can be supported by a two-part solution, where (1) full fine-grained provenance of access is exposed, enabling debugging, auditing and explanation; and (2) access to facts is computed incrementally under policy updates. Finally, our framework does not currently implement a concurrency control mechanism. It would be interesting to explore the interaction between concurrency and distributed access control, and we leave this to future work.

9. ACKNOWLEDGEMENTS

We thank Emilien Antoine for his help with the WebdamLog system [8] and Peter Alvaro for his help incorporating TCP into Bud messaging. This work has been supported in part by the ERC grant Webdam (226513). Miklau was supported in part by NSF CNS-1012748.

10. REFERENCES

[1] M. Abadi, M. Burrows, B. Lampson, and G. Plotkin. A calculus for access control in distributed systems. *ACM Trans. Program. Lang. Syst.*, 15(4):706–734, Sept. 1993.

[2] S. Abiteboul, M. Bienvenu, A. Galland, and E. Antoine. A rule-based language for Web data management. In *PODS*, 2011.

[3] S. Abiteboul, A. Galland, and N. Polyzotis. A model for web information management with access control. In *WebDB Workshop*, 2011.

[4] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[5] P. Alvaro, N. Conway, J. Hellerstein, and W. R. Marczak. Consistency analysis in bloom: a calm and collected approach. In *CIDR*, pages 249–260, 2011.

[6] P. Alvaro, W. R. Marczak, N. Conway, J. M. Hellerstein, D. Maier, and R. C. Sears. Dedalus: Datalog in Time and Space. Technical Report UCB/EECS-2009-173, EECS Department, University of California, Berkeley, December 2009.

[7] Y. Amsterdamer, D. Deutch, and V. Tannen. Provenance for aggregate queries. In *Principles of Database Systems (PODS)*, 2011.

[8] E. Antoine. *Distributed data management with the rule-based language: WebdamLog*. PhD thesis, ENS Cachan, 2013.

[9] A. Bonner. Transaction datalog: A compositional language for transaction programming. In *In Proceedings of the International Workshop on Database Programming Languages, Estes Park*. Springer, 1997.

[10] B. Carminati, E. Ferrari, R. Heatherly, M. Kantarcioglu, and B. Thuraisingham. A semantic web based framework for social network access control. In *Symposium on Access Control Models and Technologies (SACMAT)*, pages 177–186, 2009.

[11] B. Carminati, E. Ferrari, R. Heatherly, M. Kantarcioglu, and B. Thuraisingham. Semantic web-based social network access control. *Computers & security*, 30(2):108–115, 2011.

[12] Y. Cheng, J. Park, and R. Sandhu. Relationship-based access control for online social networks: Beyond user-to-user relationships. In *Privacy, Security, Risk and Trust (PASSAT), 2012 International Conference on and 2012 International Confernece on Social Computing (SocialCom)*, pages 646–655. IEEE, 2012.

[13] E. Ferrari. *Access Control in Data Management Systems*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2010.

[14] T. J. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In *PODS*, pages 31–40, 2007.

[15] H. Hu, G.-J. Ahn, and J. Jorgensen. Multiparty access control for online social networks: model and mechanisms. *Knowledge and Data Engineering, IEEE Transactions on*, 25(7):1614–1627, 2013.

[16] K. LeFevre, R. Agrawal, V. Ercegovac, R. Ramakrishnan, Y. Xu, and D. DeWitt. Limiting disclosure in hippocratic databases. In *VLDB '04: Proceedings of the Thirtieth international conference on Very large data bases*, pages 108–119. VLDB Endowment, 2004.

[17] L. E. Olson, C. A. Gunter, and P. Madhusudan. A formal framework for reflective database access control policies. In *ACM Conference on Computer and Communications Security*, pages 289–298, 2008.

[18] M. T. Özsu and P. Valduriez. *Principles of Distributed Database Systems, Third Edition*. Springer, 2011.

[19] J. Park, D. Nguyen, and R. Sandhu. A provenance-based access control model. In *Privacy, Security and Trust (PST), 2012 Tenth Annual International Conference on*, pages 137–144, 2012.

[20] B. O. O. M. project. Bloom programming language. `http://www.bloom-lang.net/`.

[21] S. Rizvi, A. Mendelzon, S. Sudarshan, and P. Roy. Extending query rewriting techniques for fine-grained access control. In *Conference on Management of Data (SIGMOD)*, pages 551–562, 2004.

[22] M. Shehab, A. Squicciarini, G.-J. Ahn, and I. Kokkinou. Access control for online social networks third party applications. *computers & security*, 31(8):897–911, 2012.

[23] Q. Wang, T. Yu, N. Li, J. Lobo, E. Bertino, K. Irwin, and J.-W. Byun. On the correctness criteria of fine-grained access control in relational databases. In *Conference on Very Large Data Bases*, pages 555–566, 2007.

APPENDIX

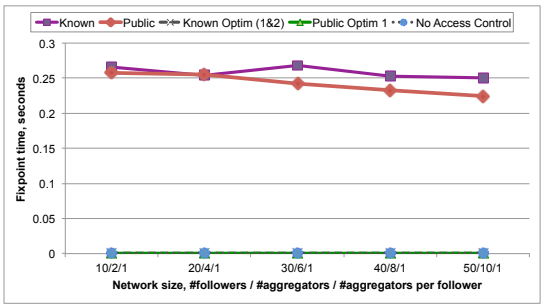
Plots and discussion in this section complement experimental results presented in Section 6.

Figures 6 and Figure 7 complement Figure 4, in which we presented fixpoint time with and without access control for the MAF-UoJ scenario as a function of network size. In Figure 6 we show fixpoint time for master and agg for the MAF-JoU scenario, and observe similar trends as for MAF-UoJ. In Figure 7, we show total running time for master, agg, and fol peers. We observe the same trends in performance for total time as for fixpoint time, with Optim (1 & 2) exhibiting comparable performance for the KNOWN policy as Optim 1 for PUBLIC. The overhead of the optimized implementation of access control is either linear or, better yet, fixed compared to no access control.

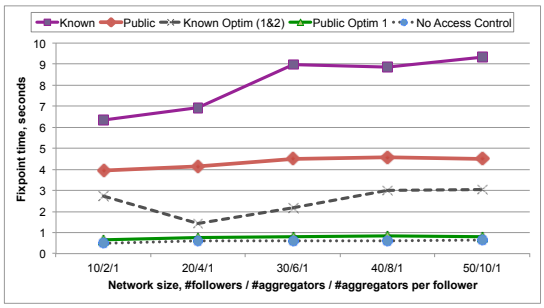
Consider Figures 7(b) and 7(d). Observe that (i) there is no increase in total time with No Access Control as network size increases, because the aggregator / follower ratio is being help constant; and (ii) that optimized versions of access control introduce a fixed overhead over the No Access Control case, despite the increasing size of the p-sets. The latter observation points to the effectiveness of Optim 2.

Figure 8 presents fixpoint and total times on peer friend in the PA scenario, averaged across peers. This figure complements Figure 5 in Section 6. (Recall from Section 6 that we were unable to run unoptimized versions of PA, and so we do not include these here.) Both PUBLIC and KNOWN are efficient and show a linear increase over No Access Control. We would expect that PUBLIC runs faster than KNOWN, which is not the case here, so this case should be further investigated.

In a final experiment, we explore the effect of network topology on performance. Figure 9 presents total time on agg and fixpoint time on fol for MAF-UoJ. These plots are representative of other results for both MAF-UoJ and MAF-JoU scenarios. We hold the size of the network constant at

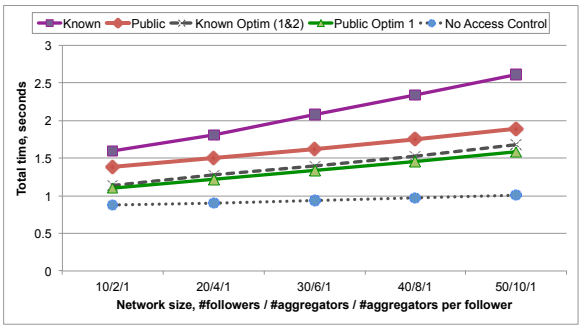


(a) MAF-JoU, time at master

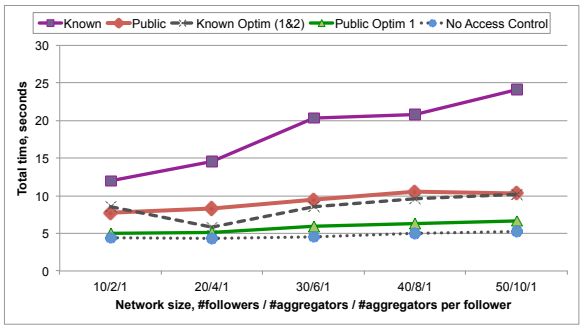


(b) MAF-JoU, average time at agg peers

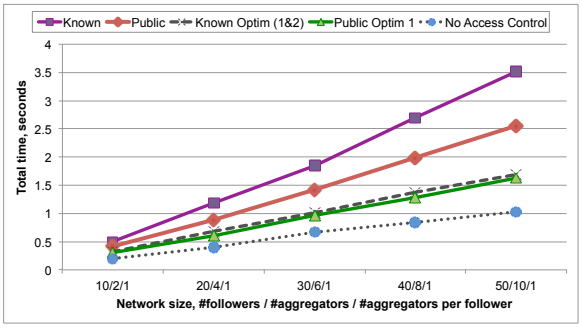
Figure 6: Fixpoint time in MAF-JoU as a function of network size.



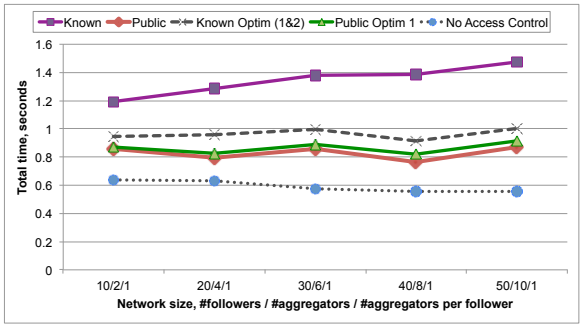
(a) MAF-JoU, time at master



(b) MAF-JoU, average time at agg peers

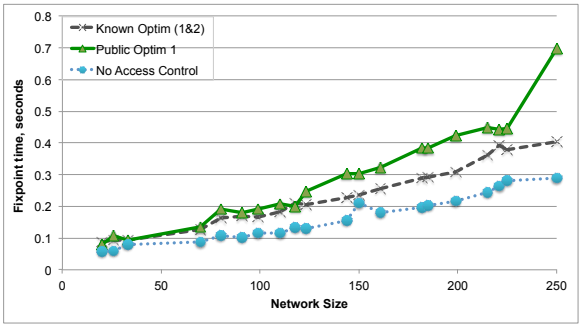


(c) MAF-UoJ, time at master

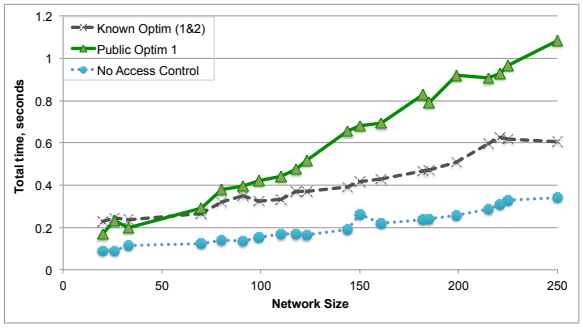


(d) MAF-UoJ, average time at fol peers

Figure 7: Total time in MAF-JoU and MAF-UoJ as a function of network size.

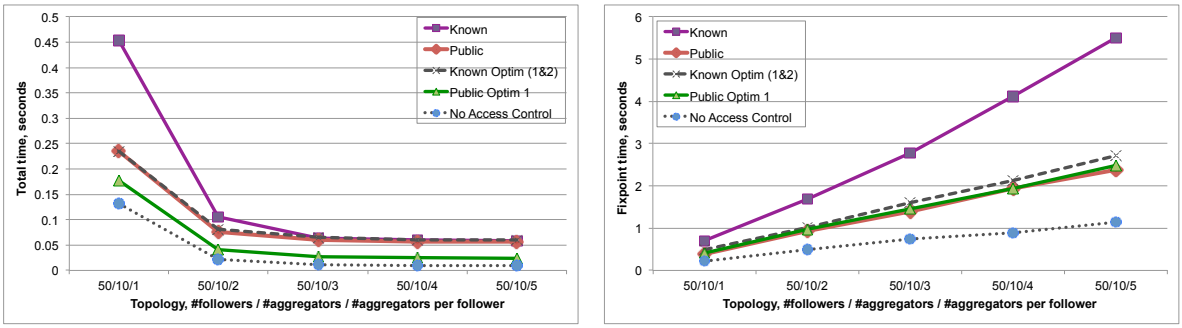


(a) fixpoint time



(b) total time

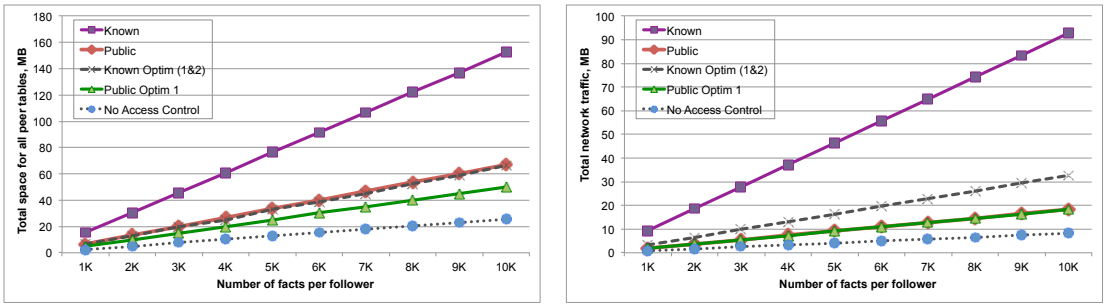
Figure 8: Running time of peer friend in PA, as a function of network size.



(a) total time, agg peer

(b) fixpoint time, fol peer

Figure 9: Running time in MAF-UoJ, as a function of the number of aggregators per follower.



(a) total space for all peers (MB)

(b) total network traffic for all peers (MB)

Figure 10: Space overhead of access control, for MAF-JoU (10/2/1).

the largest MAF size (50/10/k) and vary the number of aggregators per follower k between 1 and 5.

As the value of k increases, the number of results at an aggregator diminishes rapidly, hence the trend in Figure 9(a). We observe a significant difference between PUBLIC and KNOWN policies in unoptimized versions, which is due to the p-set growth with increasing k. Note that Optim (1 & 2) eliminates this growth successfully, with performance under the KNOWN policy matching that of the PUBLIC policy. The overhead of access control appears fixed for the agg peer and linear for the fol peer.

Figure 10(a) presents the space overhead of storing access control information: relations acl@p at each peer, p-sets annotating each intentional tuple, and any additional relations required by optimizations 1 and 2. We plot total space for all tables (both extensional and intentional, which are materialized in WebdamLog) on all peers as of the last tick. We observe that the amount of space increases linearly for all access control conditions, including no access control. Further, we observe that PUBLIC Optim 1 and KNOWN Optim (1 & 2) have lower space overhead than the respective unoptimized versions, and that this overhead is reasonable. Figure 10(b) shows the total amount of network traffic exchanged among all MAF-JoU (10/2/1) peers and we observe similar trends as in Figure 10(a): the amount of network traffic increases linearly with increasing EDB size, as expected.